Data Wrangling: Clean, Transform, Merge, Reshape

CS 3753 Data Science

Prof. Weining Zhang

Topics

- Merge and integrate data
- Reshaping and Pivoting
- Data Transformation and Preproccessing

After loading into a DataFrame, raw data needs to be pre-processed before used for data analysis

- Data from difference sources needs to be selected, combined, etc.
- Missing values needs to be filled in with other useable values
- Noise needs to be smoothed using binning methods
- Values may need to be normalized, mapped into a different range, encode/decoded, transformed into other format
- Tables may need to be merged, or with structure changed, etc.
- Pandas provide functions to perform all these tasks

In [ ]:
```
%matplotlib inline

from __future__ import division
from numpy.random import randn
import numpy as np
import os
import matplotlib.pyplot as plt
np.random.seed(12345)
plt.rc('figure', figsize=(10, 6))
from pandas import Series, DataFrame
import pandas
import pandas as pd
np.set_printoptions(precision=4, threshold=500)
pd.options.display.max_rows = 100
```

Combining and Merging Data Sets

- View DataFrames as SQL tables and join tables using merge() function
- Basic syntax:

```
leftTable.merge(rightTable, on=...,
                left_on=..., right_on=...,
                how=..., suffixes=...)
```

- Result: rows from left and right tables are merged into one row in the result table if both rows have identical values in selected merge/join columns (or indices)

Types of Relational Join

- Natural Join: Merge rows if they have identical values under columns with identical names
  - Ex: $R(A, B, C) \bowtie S(D, B, E) = RS(A, B, C, D, E)$
- Equi-Join: Merge rows if they have identical values under selected pairs of columns
  - Ex: $R(A, B, C) \bowtie_{R.A=S.D} S(D, B, E) = RS(A, B, C, D, B, E)$
- Outer joins: Keep all rows in left or right or both table, even if they do not have matching rows in the other table. Fill missing values with NaN

How Is Merge Performed

- Align rows by values in the index or selected columns
- Keep all columns from both DataFrames
- For each value, merge every row in one DataFrame with that value with every row in the other DataFrame with the same value
- If a value only appears in one DataFrame, leave NaN under the columns of the other DataFrame
- If how='inner', remove any row with NaN value

In [ ]:

```
df3 = DataFrame({'lkey': ['a', 'b', 'a', 'c', 'a'],
                 'data1': range(5)})
df4 = DataFrame({'rkey': ['a', 'a', 'b', 'b', 'd'],
                 'data2': range(5)})
```

In [ ]:

```
# Equi-join on lkey = rkey
df3.merge(df4, left_on='lkey', right_on='rkey', how='outer')
```

Exercise

What is the result from the above example if

- the columns 'lkey' and 'rkey' are renamed 'key', and the parameter on='key' is used?
- the parameter how='left' or how='right' or how='inner'is used?

Merge with Multi-Columns and Common Columns

- Can specify multiple merge/join columns in list, positions are important
- If columns common in both DataFrames are not used as the matching merge/join columns, suffix is used and can be specified to distinguish them

In [ ]:
```
left = DataFrame({'key1': ['foo', 'foo', 'bar'],
                  'key2': ['one', 'two', 'one'],
                  'lval': [1, 2, 3]})
right = DataFrame({'key3': ['foo', 'foo', 'bar', 'bar'],
                   'key2': ['one', 'one', 'one', 'two'],
                   'rval': [4, 5, 6, 7]})
```

In [ ]:
```
# outer equi-join on multiple columns
left.merge(right, left_on=['key1', 'key2'],
                  right_on=['key3', 'key2'], how='outer')
```

Exercise

Experiment with the above example by

- different ordering of the merge columns
- selection of fidderent merge columns
- specifying different suffixes

In [ ]:
```
left.merge(right, on='key2')
```

In [ ]:
```
left.merge(right, left_on='key1', right_on='key3',
           suffixes=('_left', '_right'))
```

Merging on Index

Merge can be specified by

- using the index of both DataFrames as the merge/join attribute
- using index of one DataFrame and a column of another DataFrame
- using hierarchical index
- Again, order of merge/join columns is important

In [ ]:
```
lefth = DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevad
a'],
                   'key2': [2000, 2001, 2002, 2001, 2002],
                   'data': np.arange(5.)})
righth = DataFrame(np.arange(12).reshape((6, 2)),
                   index=[['Nevada', 'Nevada', 'Ohio', 'Ohio', 'Ohio
', 'Ohio'],
                          [2001, 2000, 2000, 2000, 2001, 2002]],
                   columns=['event1', 'event2'])
lefth
```

In [ ]:
```
righth
```

In [ ]:
```
# Equi-Join on two pairs of columns
lefth.merge(righth, left_on=['key1', 'key2'],
                    right_index=True,
                    how='outer')
```

Exercise

- What would be the output if in the previous example, we specify left_on=['key2', 'key1']?
- How to specify the parameters in the previous example so that the merge is based on column 'key2' in lefth and the second level index in righth?

Using Other Functions to Perform Joins

- Universal funciton

  ```
  pd.merge(leftTable, rightTable, how, columns, ...)
  ```

- The join() function will join columns from multiple DataFrames on index or selected columns

  ```
  leftTable.join(otherTables, onCols, type..)
  ```

  - Requires common index levels or common columns

In [ ]:
```
left2 = DataFrame([[1., 2.], [3., 4.], [5., 6.]],
                  index=['a', 'c', 'e'],
                  columns=['Ohio', 'Nevada'])
right2 = DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
                   index=['b', 'c', 'd', 'e'],
                   columns=['Missouri', 'Alabama'])
```

In [ ]:
```
left2
```

In [ ]:
```
right2
```

In [ ]:
```
pd.merge(left2, right2, how='outer',
                        left_index=True,
                        right_index=True)
```

In [ ]:
```
another = DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
                    index=['a', 'c', 'e', 'f'],
                    columns=['New York', 'Oregon'])
```

In [ ]:
```
# join three tables
left2.join([right2, another])
```

In [ ]:
```
left2.join([right2, another], how='outer', sort=True)
```

Exercise

- What is the result of pd.merge([df1, df2, df3], on=col), where col is a common column in df1, df2, and df3?
- What is the result of pd.merge(df, df)?
- What is the result of pd.merge(df1, df2) where df1 and df2 do not have any common column?

Concatenation of DataFrames

- Use function

```
pd.concat(listOfTables, axis=...,
          join=..., join_axes=...,
          keys=..., ...)
```

- Also work on Series
- Similar to NumPy function np.concatenate(), which by default, put matrices side-by-side

Concatenating Along An Axis

- Patterns

```
axis=0                      axis=1

table1                      table1 table2 table3
table2
table3
```

- Keep all columns, align on index, may add a new index/column level to separate tables
- Fill NaN in missing columns or index

In [ ]:
```
# Concatenate DataFrames
df1 = DataFrame(np.arange(6).reshape(3, 2), index=['a', 'c', 'b'],
                columns=['one', 'two'])
df2 = DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
                columns=['three', 'four'])
```

In [ ]:
```
df1
```

In [ ]:
```
df2
```

In [ ]:
```
pd.concat([df1, df2], axis=1,
          keys=['level1', 'level2'],
          sort=False)
```

Exercise

What differences does it make if in the previouse example, we replace axis=1 or sort=False?

In [ ]:
```
# Can also pass input DataFrames in Dict
pd.concat({'level1': df1, 'level2': df2}, axis=1,
          sort=True)
```

In [ ]:
```
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],
          names=['upper', 'lower'], sort=False)
```

In [ ]:
```
# Concatenate by appending rows and making new index
df1 = DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])
df2 = DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])
```

In [ ]:
```
df1
```

In [ ]:
```
df2
```

In [ ]:
```
pd.concat([df1, df2], ignore_index=True,
          axis=1, keys=['level1', 'level2'],
          sort=False)
```

In [ ]:
```
# Compare np.concatenate and pd.concat
df1 = DataFrame({'a': [1,2,3],
                 'b': [2, 3, 4]})
df2 = DataFrame({'b': [4, 5, 6],
                 'c': [2, 3, 5]})
np.concatenate([df1, df2], axis=1)
```

In [ ]:
```
pd.concat([df1, df2], axis=1)
```

Exercise

Use pd.concat() function to create a DataFrame from multiple Series that represent the columns.

Combining Data with Overlap

- Two tables can be combined using
  `table1.combine_first(table2)`
  - If both tables have a value at the same cell location, the elements in table1 will be selected
  - If a cell is not in table1, it will be created with a NaN filled in

In [ ]:
```
df1 = DataFrame({'a': [1., np.nan, 5., np.nan],
                  'b': [np.nan, 2., np.nan, 6.],
                  'c': range(2, 18, 4)})
df2 = DataFrame({'a': [5., 4., np.nan, 3., 7.],
                  'b': [np.nan, 3., 4., 6., 8.]})
df1.combine_first(df2)
```

Reshaping and Pivoting

- Use the stack() and unstack() functions to change the shape and appearance of DataFrame
  - Especially useful with hierarchical index
- Use pivot() function to change between long and wide table formats

Reshaping with Hierarchical Indexing

Table can be reshaped by converting a level of index into a level of column labels and vice versa.

```
   table.unstack(indexLevel): making index labels in that level the
lowest level of column labels

   table.stack(columnLabelLevel): making the colmn labels at that l
evel the lowest level of index
```

In [ ]:
```
data = DataFrame(np.arange(6).reshape((2, 3)),
                 index=pd.Index(['Ohio', 'Colorado'], name='state'),
                 columns=pd.Index(['one', 'two', 'three'], name='num
ber'))
data
```

In [ ]:
```
result = data.stack()
result
```

In [ ]:
```
result.unstack()
```

In [ ]:
```
result.unstack(0)
```

In [ ]:
```
result.unstack('state')
```

In [ ]:
```
df = DataFrame({'left': result, 'right': result + 5},
               columns=pd.Index(['left', 'right'], name='side'))
df
```

In [ ]:
```
df.unstack('state')
```

In [ ]:
```
df.unstack('state').stack('side')
```

The Idea of Pivoting

- A tables columns can be separated into dimensions and values. Such a table is in the Long format.
- The table can be reorganized to have columns labeled by the values in some dimentional columns, index labeled by remaining dimensional columns. This is called a Wide format of the table.
- The cells are all from the value columns

```
pd.pivot(columnForIndex, columnForColumns, columnsOfValues
)
```

In [ ]:
```
data = pd.read_csv('ch07/macrodata.csv')
periods = pd.PeriodIndex(year=data.year, quarter=data.quarter, name=
'date')
data = DataFrame(data.to_records(),
                 columns=pd.Index(['realgdp', 'infl', 'unemp'], name
='item'),
                 index=periods.to_timestamp('D', 'end'))
data[:10]
```

In [ ]:
```
# the long format
ldata = data.stack().reset_index().rename(columns={0: 'value'})
ldata[:10]
```

In [ ]:
```
# the wide format
# wdata = ldata.pivot('date', 'item', 'value')
# wdata[:10]
```

In [ ]:
```
# The Wide Table Format
# Use date as index, items as columns, values as elements
pivoted = ldata.pivot('date', 'item', 'value')
pivoted.head()
```

In [ ]:
```
# Add one more column of values
ldata['value2'] = np.random.randn(len(ldata))
ldata[:10]
```

In [ ]:
```
pivoted = ldata.pivot('date', 'item')
pivoted[:5]
```

In [ ]:
```
# Equivalently
unstacked = ldata.set_index(['date', 'item']).unstack('item')
unstacked[:7]
```

In [ ]:
```
# Show only one column of value
pivoted['value'][:5]
```

Data Transformation

- Remove duplicate data
- Change data values by applying function
- Normalization
- Discritizaiton and Noise Smoothing
- Filling missing values
- Detect and replace outliers
- Data reduction using random sampling

Removing Duplicate Data

```
- Locate duplicate values:  df.duplicated()
- remove duplicate values:  df.drop_duplicates()
```

In [ ]:
```
data = DataFrame({'k1': ['one'] * 3 + ['two'] * 4,
                  'k2': [1, 1, 2, 3, 3, 4, 4]})
data
```

In [ ]:
```
data.duplicated()
```

In [ ]:
```
data.drop_duplicates()
```

In [ ]:
```
data['v1'] = range(7)
print(data)
data.drop_duplicates(['k1'])
```

In [ ]:
```
data.drop_duplicates(['k1', 'k2'], keep='last')
```

Transforming Data Using a Function or a Mapping

One or more function can be applied to every element (or cell) in any slice of a DataFrame.

```
df.map(<function>)
```

- Apply the function to each element in a Series
- To apply a funciton to an entire row or column, use `df.apply(<function>)`

In [ ]:
```
data = DataFrame({'food': ['bacon', 'pulled pork', 'bacon', 'Pastram
i',
                           'corned beef', 'BACON', 'pastrami', 'hone
y ham',
                           'nova lox'],
                  'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
data
```

In [ ]:
```
# Want to add a column identify type of animal
# from which the type of meat is made
meat_to_animal = {
  'bacon': 'pig',
  'pulled pork': 'pig',
  'pastrami': 'cow',
  'corned beef': 'cow',
  'honey ham': 'pig',
  'nova lox': 'salmon'
}
```

In [ ]:
```
# map applies a function to each element in the series
data['animal'] = data['food'].map(str.lower).map(meat_to_animal)
data
```

In [ ]:
```
# Apply a lambda function defined at the call
data['food'].map(lambda x: meat_to_animal[x.lower()])
```

Exercise

Given a DataFrame df with only numerical values, apply a function to df to return a Series M, where for each row i of df, M[i] is the maximum value in row i in df

Normalization

Convert values from an arbitrary domain into a fixed domain, ex. [0, 1], or [-2, 2].

- MiniMax Normalization: from $[min_A, max_A]$ to $[newmin_A, newmax_A]$
$$v' = \frac{v - min_A}{max_A - min_A}(newmax_A - newmin_A) + newmin_A$$
- Z-score Normalization: (μ: mean, σ: standard deviation):
$$v' = \frac{v - \mu_A}{\sigma_A}$$
- Decimal scaling:
$$v' = \frac{v}{10^j}$$
where j is the smallest integer such that $\max(|v'|) < 1$

In [ ]:
```
a = np.random.rand(20)
print("a =\n", a)
# MiniMax Normalization into range [3, 10]
m = a.min()
M = a.max()
b = ((a-m)/(M-m))*(10-3)+3
print("b =\n", b)
# Z-score Normalization
mu = a.mean()
sd = a.std()
c = (a-mu)/sd
print("c =\n", c)
```

Exercise

Define a function which takes a DataFrame df and a column name col as parameters and applies the decimal scaling normalization to df[col].

Replacing Values and Fill in Missing Values

It may be neccessary to replace some outlier or missing values by other values. For this purpose, Pandas provides

```
replace(listOfValues, listOfNewValues)
fillna()
```

In [ ]:
```
data = Series([1., -999., 2., -999., -1000., 3.])
data
```

In [ ]:
```
data.replace(-999, np.nan)
```

In [ ]:
```
data.replace([-999, -1000], np.nan)
```

In [ ]:
```
data.replace([-999, -1000], [np.nan, 0])
```

In [ ]:
```
# Replacing based on a dict
data.replace({-999: np.nan, -1000: 0})
```

Filling Missing Values

In [ ]:
```
df = pd.DataFrame({'id': range(10),
                   'a' : [1.0, 3.0, np.nan, 4.0, 5.0, np.nan, np.nan
, 4.0, 5.0, 3.0],
                   'b' : ['red', None, 'blue', 'blue', None, None, '
gree', 'black', 'red', 'red']})
print(df)
df['a'] = df.a.fillna(df.a.mean())
df['b'] = df.b.fillna(df.b.mode()[0])
print(df)
```

Renaming Axis Indexes

```
df.index.map(function): apply function to row labels
df.rename(index=func1, columns=func2)
```

- renaming can be specified by functions or by dicts

In [ ]:
```
data = DataFrame(np.arange(12).reshape((3, 4)),
                 index=['Ohio', 'Colorado', 'New York'],
                 columns=['one', 'two', 'three', 'four'])
data
```

In [ ]:
```
data.index.map(str.upper)
```

In [ ]:
```
data.index = data.index.map(str.upper)
data
```

In [ ]:
```
# Change index and column case
data.rename(index=str.title, columns=str.upper)
```

In [ ]:
```
# Change index and column names
data.rename(index={'OHIO': 'INDIANA'},
            columns={'three': 'peekaboo'})
```

In [ ]:
```
# Always returns a reference to a DataFrame
_ = data.rename(index={'OHIO': 'INDIANA'}, inplace=True)
data
```

Discretization and Noise Smoothing with Binning

- Discretization is to convert numeric data into categorical.
- Smoothing is to remove noise or extreem values
- Binning is a common method for discretization. Sort data values and place values into bins, then replace the values by its bin labels
- Binning can also be used to smooth data to reduce noises. For example, after binning, values in a column can be replaced by bin means or bin median, so that noises or outliers are smoothed out.
- Pandas provides two functions cut() and qcut() to perform binning

Equi-Width Binning Using cut()

- **Equi-width Binning**: Divide the data range [min, max] equally into n sub-ranges. Assign each data to the bin according to its subrange. For example, we can divide the range (0, 100] into 4 bins: (0, 25], (25, 50], (50, 75], (75, 100]. The width of each bin is 25.
- cut(data, bins, labels): binning by range of value, can do equi-width binning
- For each input data, the output keeps the bin (either the label or the boundaries) for that data. The output is an object, also keeps the intervals of the bins

In [ ]:
```
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

In [ ]:
```
# Set left/right boundaries of bins, and place data into bins
bins = [18, 25, 35, 60, 100]
cats = pd.cut(ages, bins)
cats
```

In [ ]:
```
# Find the bin id for each data value
cats.codes
```

In [ ]:
```
cats.describe
```

In [ ]:
```
cats.ravel
```

In [ ]:
```
pd.value_counts(cats)
```

In [ ]:
```
pd.cut(ages, [18, 26, 36, 61, 100], right=False)
```

In [ ]:
```
# assign different names to bins
group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
pd.cut(ages, bins, labels=group_names)
```

In [ ]:
```
# equal-width bins with bin size automatically calculated from min/m
ax values
data = np.random.rand(20)
print(data)
pd.cut(data, 4, precision=2)
```

Equi-Depth Binning Using qcut()

- **Equi-depth Binning**: Divide the data range into bins of different width, so that, the bins contain the same number of data. For example, suppose there are 1000 data items. We can divide the range into 4 bins, so that, each bin contains 250 data items.
- qcut(data, quantile, labels): binning by quantile of the rank, can do equi-depth binning

In [ ]:
```
# Equal-depth bins where bins have equal numbers of values
data = np.random.randn(1000) # Normally distributed
cats = pd.qcut(data, 4) # Cut into quartiles
cats
```

In [ ]:
```
pd.value_counts(cats)
```

In [ ]:
```
pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
```

In [ ]:
```
pd.value_counts(pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.]))
```

Exercise

Replace each input value by the mean of its bin

Detecting and Filtering Outliers

- Identify outlier (or extreem values)
  - check statistics
  - view boxplot
  - test using threshold values
- Once identified, outliers can be either removed or replaced

In [ ]:
```
np.random.seed(12345)
data = DataFrame(np.random.randn(1000, 4))
data.describe()
```

In [ ]:
```
data
```

In [ ]:
```
# Find in column 3, values with absolute value greater than 3
col = data[3]
col[np.abs(col) > 3]
```

In [ ]:
```
np.abs(data) > 3
```

In [ ]:
```
(np.abs(data) > 3).any(1)
```

In [ ]:
```
# Select rows that contains a value >3 or <-3
data[(np.abs(data) > 3).any(1)]
```

In [ ]:
```
# Cap the values by -3 and +3
data[np.abs(data) > 3] = np.sign(data) * 3
data.describe()
```

In [ ]:
```
data[(np.abs(data) == 3).any(1)]
```

Permutation and Random Sampling

- One way to reduce data volume is by random sampling.
- Pandas provides several functions to take random samples
  - permutaiton() randomly reorder values in a series or rows in a DataFrame. This function can be used together with take() function to get random samples from DataFrames.
  - another function is sample()
- There are several ways to take random samples.
  - **Random sampling without replacement**: Take n random samples, and no two samples can be same.
  - **Random sampling with replacement**: Take n samples randomly and allow the same sample to be selected multiple times.

In [ ]:
```
df = DataFrame(np.arange(5 * 4).reshape((5, 4)))
df
```

In [ ]:
```
# Get a permutaion of [0, 1, 2, 3, 4]
sampler = np.random.permutation(5)
sampler
```

In [ ]:
```
# take rows from a DataFrame in the given order
df.take(sampler)
```

In [ ]:
```
# take randomly selected 3 rows without replacement
df.take(np.random.permutation(len(df))[:3])
```

In [ ]:
```
# or use the DataFrame.sample() function
df.sample(n=3)
```

In [ ]:
```
# take a random sample of 10 with replacement
bag = np.array([5, 7, -1, 6, 4])
bag
```

In [ ]:
```
sampler = np.random.randint(0, len(bag), size=10)
sampler
```

In [ ]:
```
draws = bag.take(sampler)
draws
```

In [ ]:
```
# Another example
df.sample(n=3, replace=True)
```

Additional Topics

Computing Indicator / Dummy Variables

In [ ]:
```
df = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
                'data1': range(6)})
df
```

In [ ]:
```
# Which row contains a, b, c in column key?
pd.get_dummies(df['key'])
```

In [ ]:
```
dummies = pd.get_dummies(df['key'], prefix='key')
df_with_dummy = df[['data1']].join(dummies)
df_with_dummy
```

In [ ]:
```
mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('../week02/ch02/movielens/movies.dat', sep=':
:', header=None,
                        names=mnames)
movies[:10]
```

In [ ]:
```
genre_iter = (set(x.split('|')) for x in movies.genres)
genres = sorted(set.union(*genre_iter))
genres
```

In [ ]:
```
# Create a table filled with zeros for all genres
dummies = DataFrame(np.zeros((len(movies), len(genres))), columns=ge
nres)
dummies
```

In [ ]:
```
# for each movie, set each genres to 1 in the dummies table
for i, gen in enumerate(movies.genres):
    dummies.ix[i, gen.split('|')] = 1
dummies
```

In [ ]:
```
movies_windic = movies.join(dummies.add_prefix('Genre_'))
movies_windic.iloc[:3]
```

In [ ]:
```
# Combine get_dummies() and cut() functions to get an statistical in
dicator for data
np.random.seed(12345)
```

In [ ]:
```
values = np.random.rand(10)
values
```

In [ ]:
```
bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
pd.get_dummies(pd.cut(values, bins))
```

String Manipulation

- Use string object methods
- Use regular expression package

String Object Methods

```
s.split(<delimiter>)
s.join(<listOfStrings>)
s.index(<subString>)
s.find(<substring>)
s.replace(<substring>, <replacement>)
```

In [ ]:
```
val = 'a,b,  guido'
val.split(',')
```

In [ ]:
```
pieces = [x.strip() for x in val.split(',')]
pieces
```

In [ ]:
```
first, second, third = pieces
first + '::' + second + '::' + third
```

In [ ]:
```
'::'.join(pieces)
```

In [ ]:
```
'guido' in val
```

In [ ]:
```
val.index(',')
```

In [ ]:
```
val.find(':')
```

In [ ]:
```
# index() is like find(), but raise ValueError when the substring is
not found.
# val.index(':')
```

In [ ]:
```
val.count(',')
```

In [ ]:
```
val.replace(',', '::')
```

In [ ]:
```
val.replace(',', '')
```

Regular Expressions

```
import re
re.compile(<pattern>) :  define a search pattern
re.split(p, text)     :  split text into lex tokens
patt.findall(text)    :  find occurrence of pattern in text
patt.search(text)     :  return found patterns
```

In [ ]:
```
import re
text = "foo    bar\t baz  \tqux"
re.split('\s+', text)
```

In [ ]:
```
# compile a regular expression as a pattern
regex = re.compile('\s+')
regex.split(text)
```

In [ ]:
```
regex.findall(text)
```

In [ ]:
```
text = """
Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""
```

In [ ]:
```
# Define a pattern for email address
pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

# re.IGNORECASE makes the regex case-insensitive
regex = re.compile(pattern, flags=re.IGNORECASE)
```

In [ ]:
```
regex.findall(text)
```

In [ ]:
```
# search returns the first match any where in a line
m = regex.search(text)
m
```

In [ ]:
```
text[m.start():m.end()]
```

In [ ]:
```
# Match only matches from the beginning of the text, nowhere else
print(regex.match(text))
```

In [ ]:
```
# sub() replace the first group by the string in the text
print(regex.sub('REDACTED', text))
```

Group Sub-Patterns to Extract Data

- Specify sub-pattern in (..)

```
  r'(group1)...(group2)...(groupk)'

  patt.findall(text)  :   return groups for each found ins
tance

  patt.sub(sub-pattern, text) :  further fomat groups
```

In [ ]:
```
pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'
regex = re.compile(pattern, flags=re.IGNORECASE)
```

In [ ]:
```
m = regex.match('wesm@bright.net')
m.groups()
```

In [ ]:
```
# returns list of tuples representing the matched groups
regex.findall(text)
```

In [ ]:
```
print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))
```

In [ ]:
```
# Assign names to pattern groups
regex = re.compile(r"""
    (?P<username>[A-Z0-9._%+-]+)
    @
    (?P<domain>[A-Z0-9.-]+)
    \.
    (?P<suffix>[A-Z]{2,4})""", flags=re.IGNORECASE|re.VERBOSE)
```

In [ ]:
```
m = regex.match('wesm@bright.net')
m.groupdict()
```

Vectorized String Functions

Pandas Series uses a sub() function to apply string operations on each data item, and can deal with NA values

In [ ]:
```
data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
        'Rob': 'rob@gmail.com', 'Wes': np.nan}
data = Series(data)
```

In [ ]:
```
data
```

In [ ]:
```
data.isnull()
```

In [ ]:
```
data.str.contains('gmail')
```

In [ ]:
```
pattern
```

In [ ]:
```
data.str.findall(pattern, flags=re.IGNORECASE)
```

In [ ]:
```
matches = data.str.match(pattern, flags=re.IGNORECASE)
matches
```

In [ ]:
```
matches.str.get(1)
```

In [ ]:
```
matches.str[0]
```

In [ ]:
```
data.str[:5]
```

Example: USDA Food Database

{ "id": 21441, "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA CRISPY, Wing, meat and skin with breading", "tags": ["KFC"], "manufacturer": "Kentucky Fried Chicken", "group": "Fast Foods", "portions": [{ "amount": 1, "unit": "wing, with skin", "grams": 68.0 }, ...], "nutrients": [{ "value": 20.8, "units": "g", "description": "Protein", "group": "Composition" }, ...] }

Each food has a number of identifying attributes along with two lists of nutrients and portion sizes. Having the data in this form is not particularly amenable for analysis, so we need to do some work to wrangle the data into a better form.

In [ ]:
```
import json
db = json.load(open('ch07/foods-2011-10-03.json'))
len(db)
```

In [ ]:
```
db[:2]
```

In [ ]:
```
# db[0] is the entire file
db[0].keys()
```

In [ ]:
```
db[0]['nutrients'][0]
```

In [ ]:
```
# Load the nutrients of the first food into a DataFrame
nutrients = DataFrame(db[0]['nutrients'])
nutrients[:7]
```

In [ ]:
```
# Load some remaining info into another DataFrame
info_keys = ['description', 'group', 'id', 'manufacturer']
info = DataFrame(db, columns=info_keys)
```

In [ ]:
```
info[:5]
```

In [ ]:
```
info
```

In [ ]:
```
# Counting foods by their food groups
pd.value_counts(info.group)[:10]
```

In [ ]:
```
# Build a long table of nutrients for all foods
nutrients = []

# Build a DataFrame of nutrients for each food
for rec in db:
    fnuts = DataFrame(rec['nutrients'])
    fnuts['id'] = rec['id']
    nutrients.append(fnuts)

# Concatenate into a long table
nutrients = pd.concat(nutrients, ignore_index=True)
```

In [ ]:
```
nutrients
```

In [ ]:
```
# Drop duplicate
nutrients.duplicated().sum()
```

In [ ]:
```
nutrients = nutrients.drop_duplicates()
```

In [ ]:
```
# Renaming columns to make specific meanings of each column
col_mapping = {'description' : 'food',
               'group'       : 'fgroup'}
info = info.rename(columns=col_mapping, copy=False)
info
```

In [ ]:
```
col_mapping = {'description' : 'nutrient',
               'group' : 'nutgroup'}
nutrients = nutrients.rename(columns=col_mapping, copy=False)
nutrients
```

In [ ]:
```
# join the two tables
ndata = pd.merge(nutrients, info, on='id', how='outer')
```

In [ ]:
```
ndata
```

In [ ]:
```
ndata.ix[30000]
```

In [ ]:

```
# Plot the histogram of food groups
result = ndata.groupby(['nutrient', 'fgroup'])['value'].quantile(0.5
)
result['Zinc, Zn'].sort_values().plot(kind='barh')
```

In [ ]:

```
# Find which food is most dense in each nutrient
by_nutrient = ndata.groupby(['nutgroup', 'nutrient'])

get_maximum = lambda x: x.xs(x.value.idxmax())
get_minimum = lambda x: x.xs(x.value.idxmin())

max_foods = by_nutrient.apply(get_maximum)[['value', 'food']]

# make the food a little smaller
max_foods.food = max_foods.food.str[:50]
```

In [ ]:

```
# Example for Amino Acids
max_foods.loc['Amino Acids']['food']
```